Multiplication in Three Pieces: 20 pts. In class, we saw a divide and conquer algorithm for multiplication that divided each $n$ bit integer into high and low positions, each $n/2$ bits long. Consider algorithms that break the integers up into three pieces instead, the high order, mid order, and low order pieces, each $n/3$ bits long. What is the best divide-and-conquer multiplication algorithm of this type you can find? Is it better or worse than the two piece algorithm from class?

Assume that $x$ and $y$ are $n$ digit numbers, and, assume for convenience, that $n$ is divisible by 3. Then we can write $x = x_210^{2n/3} + x_110^{n/3} + x_0$, and $y = y_210^{2n/3} + y_110^{n/3} + y_0$, where each $x_i, y_i$ is a block of consecutive digits in $x$ or $y$. Then multiplying these expressions gives: $xy = x_2y_210^{4n/3} + (x_2y_1 + x_1y_2)10^n + (x_2y_0 + x_1y_1 + x_0y_2)10^{2n/3} + (x_1y_0 + y_0x_1)10^{n/3} + x_0y_0$. Let the co-efficient of $10^{2n/3}$ be called $e_3$, i.e., $e_0 = x_0y_0, e_1 = x_1y_0 + x_0y_1, e_2 = e_1y_0 + x_2y_1 + x_1y_2, e_3 = e_2y_0 + x_2y_1 + x_1y_2$, etc. If we do this recursion in the obvious way, we have to do 9 recursive calls to multiply numbers with $n/3$ digits, for a recurrence relation $T(n) = 9T(n/3) + cn$ for some constant $c$. In the general format for divide-and-conquer recurrences, we have $A = 9, B = 3, C = 1$, so since $A/B^C > 1$, the time is $O(n \log A/\log B) = O(n \log 9/\log 3) = O(n^2)$, which is no improvement over the grade-school method or obvious divide-and-conquer in blocks of 2.

However, we can compute the terms in a clever way like we did in pairs: compute $a = x_0y_0, b = x_1y_1, c = x_2y_2, d = (x_0 + x_1)(y_0 + y_1), e = (x_0 + x_2)(y_0 + y_2)$ and $f = (x_2 + x_0)(y_2 + y_0)$. Then $e_0 = a, e_1 = d - a - b, e_2 = e - a - c + b, e_3 = f - a - c$, and $e_4 = c$. So we get a recurrence of approximately the form: $T(n) = 6T(n/3) + O(n)$. Which gives us the time $T(n) = O(n^{1.63})$, which is worse than when we did the clever multiplication in pairs. So, as almost everyone concluded, it is better to do divide-and-conquer multiplication in pairs than to do it this way.

BUT we can be REALLY tricky. Consider the polynomials $p_x(z) = x_2z^2 + x_1z + x_0$ and $p_y(z) = y_2z^2 + y_1z + y_0$. Then $p_x(z)p_y(z) = e_4z^4 + e_3z^3 + e_2z^2 + e_1z + e_0$ is a polynomial of degree 4. We can determine a polynomial of degree 4 by its values at 5 points: pick for simplicity $z = -2, -1, 0, 1, 2$. This suggests doing the following computations: $a = (4x_2 - 2x_1 + x_0)(4y_2 - 2y_1 + y_0) = 16e_4 - 8e_3 + 4e_2 - 2e_1 + e_0, b = (x_2 - x_1 + x_0)(y_2 - y_1 + y_0) = e_4 - e_3 + e_2 - e_1 + e_0, c = x_0y_0, d = (x_2 + x_1 + x_0)(y_2 + y_1 + y_0) = e_4 + e_3 + e_2 + e_1 + e_0, e = (4x_2 + 2x_1 + x_0)(4y_2 + 2y_1 + y_0) = 16e_4 + 8e_3 + 4e_2 + 2e_1 + e_0$. Then we get back $e_4, \ldots, e_0$ as follows:
1. \( e_0 = c \),
2. \( f = (d - b)/2 = e_1 + e + 3 \),
3. \( g = (e - a)/2 = 8e_3 + e_1 \)
4. \( e_3 = (g - f)/7 \),
5. \( e_1 = f - e_3 \),
6. \( h = (b + d)/2 - e_0 = e_4 + e_2 \)
7. \( i = (a + c - 2e_0)/8 = 4e_4 + e_2 \),
8. \( e_4 = (i - h)/3 \),
9. \( e_2 = h - e_4 \).

Then we get a recursion: \( T(n) = 5T(n/3) + O(n) \), which comes out to \( O(n^{\log_5/\log_3}) = O(n^{1.51}) \) which is better than in twos (for REALLY big \( n \).)

This method generalizes, to get theoretically better and better algorithms as the number of pieces increases. The limit of this seems to be the Fast Fourier Multiplication Algorithm, which multiplies in time \( O(n\log^2 n) \). The secret of the Fast Fourier is to pick the points at which to evaluate the polynomial carefully, to allow much of the work for different points to be identical.

Base conversion: 20 points Consider the problem of converting a base 3 integer into decimal. Give an efficient algorithm (in terms of single digit operations) for this problem, trying to beat the \( O(n^2) \) algorithm on the calibration homework. You can use the multiplication algorithm from class (or above) as a subroutine.

The following recursive algorithm uses the divide and conquer method to convert an \( n \) bit binary integer \( x_{n-1}...x_0 \) into decimal. It uses the \( O(n^{\log_2 3}) \) time divide-and-conquer multiplication algorithm \( \text{Multiply2} \) from class and the text; and the grade school linear time \( O(n) \) \( \text{Add} \) algorithm as sub-routines. We assume Add and Multiply are defined to take decimal integers as input and output. Note that 2\( n \), in binary, is a 1 followed by \( n \) 0's, so is easy to construct as a binary integer in linear time. Let \( \text{ConstructPower2} \), given \( n \), construct 2\( n \) in binary in time \( O(n) \).

\( \text{ConvertToBinary}(x_{n-1}...x_0) \): Binary integer represented as an array of bits; decimal integer;

1. IF \( n = 1 \) return \( x_0 \).
2. \( y \leftarrow x_{n-1}...x_{n/2} \)
3. \( z \leftarrow x_{n/2-1}...x_0 \)
4. \( w \leftarrow \text{ConstructPower2}(n/2 - 1) \) (in binary)
5. $a \leftarrow \text{ConvertToBinary}(y)$
6. $b \leftarrow \text{ConvertToBinary}(z)$
7. $c \leftarrow \text{ConvertToBinary}(w)$
8. $c \leftarrow \text{Add}(c, c)$
9. $d \leftarrow \text{Multiply}(2(a, c))$
10. $e \leftarrow \text{Add}(d, b)$
11. Return $e$

For the time analysis, we make three recursive calls, on $w, y$ and $z$. Now, $w, y, z$ are all $n/2$ bit binary integers. The time to construct them is $O(n)$. The results are decimal versions, and so have fewer digits (by about a log 10 factor). Thus $a, b, c, d$ are at most $O(n)$ digits each, so the time for the two Adds is $O(n)$ and the Multiply is $O(n \log^3)$, SO the total time out of the recursion is $O(n \log^3)$.

This gives $T(n) = 3T(n/2) + O(n \log^3)$ as the recurrence. This meets the format of Theorem B.5 with $A=3$, $B=2$, $K = \log 3$. Then since $3 = 2^{\log 3}$, we are in the steady-state case, so $T(n) \in O(n \log^3 \log n)$.

I don’t know if you can do better iteratively. If you try the straight-forward approach, it goes something like:

1. $\text{Convert}(x_{n-1..x_0})$;
2. $\text{Powers}[0] = 1$;
3. For $I=1$ to $n$ DO: $\text{Powers}[I] = \text{Add}([\text{Powers}[I - 1], \text{Powers}[I - 1]])$; {this computes the powers of 2 in decimal and stores them in an array}
4. $X = 0$.
5. For $I = 1$ to $N$ do:
6. IF $x_I = 1$ THEN $X \leftarrow \text{Add}(X, \text{Powers}[I])$;
7. Return $X$

Since each Add is $O(n)$, this algorithm is $O(n^2)$. So our divide-and-conquer approach is better.

One thing we could do better is observe that for $n = 2^k$, all the powers of 2 we use in the divide and conquer are actually for $2^i$, $0 \leq i \leq \log n$. So we could pre-compute all of these using $\text{Exponent}[i] = \text{Multiply}([\text{Exponent}(i - 1), \text{Exponent}(i - 1)])$ and then replace the recursive call to get $c$ by setting $c$ to be the precomputed element. This will remove the $\log n$ factor from the order, which isn’t much. However, if we improve the Multiply algorithm, we also get the improvement in this algorithm, unlike the one we did first. So the limit might be using Fast-Fourrier Multiplication in the modified algorithm sketched above.
**Least Common Ancestor: 20 points** Consider the following recursive algorithm that takes as input a binary tree $T$.

Each non-leaf in $T$, $x$, has left-child $x.left$, and right child $x.right$, and each non-root has parent $x.parent$. (Child pointers at leaves and the parent pointer at the root return NIL). It uses a depth-first search procedure $DFS$ that is linear-time in the size of the sub-tree and returns the list of nodes in the sub-tree. It computes, for each pair of nodes $x$ and $y$ in $T$, the deepest node that is an ancestor of both $x$ and $y$, and stores it in an array $LCA[x,y]$. The main idea is that if $x$ is in the left sub-tree of the root, and $y$ is in the right sub-tree, then the only common ancestor of $x$ and $y$ is the root. Otherwise, the least common ancestor is in the subtree that contains both $x$ and $y$.

LeastCommonAncestor(r: node)

1. $LCA[r,r] \leftarrow r$
2. IF $r.left \neq NIL$ THEN
3.  LeastCommonAncestor(r.left);
4.  $L_1 \leftarrow DFS(r.left)$;
5.  IF $r.right \neq NIL$ THEN
6.  LeastCommonAncestor(r.right);
7.  $L_2 \leftarrow DFS(r.right)$.
8.  FOR each $x \in L_1$
9.    FOR each $y \in L_2$
10.   $LCA[x,y] \leftarrow r$

First, give a recurrence relation for the time of this algorithm when the input is a complete binary tree of size $n = 2^d - 1$, where $d$ is the depth of the tree. (Note that such a complete binary tree is always perfectly balanced, with left and right sub-trees of the same size.), and solve it to give a time analysis for the algorithm in the complete binary tree case. Then give a worst-case analysis for the time, not making any assumptions about the input tree.

Let $l$ be the size of the left sub-tree and $r$ the size of the right subtree. In all cases, $n = l + r + 1$.

Since we make one recursive call to the left sub-tree and one to the right sub-tree, and the rest of the algorithm is dominated by the two nested for loops, that go through all $l$ vertices on the left and $r$ vertices on the right, we get $T(n) \leq T(l) + T(r) + c(l + n)$. The $O(n)$ term was added in case one of $l, r$ is 0, and we need to take the $DFS$ time into account. In the complete binary tree case, $l = r = (n - 1)/2$ and this...
Hamiltonian path

Consider the following algorithm for deciding whether a graph has a Hamiltonian Path from x to y, i.e., a simple path in the graph from x to y going through all the nodes in G exactly once. (\(N(x)\) is the set of neighbors of x, i.e. nodes directly connected to x in G).

1. \(HamPath(G, x : node, y : node)\)
2. If \(x = y\) is the only node in G return True.
3. If no node in G is connected to x, return false.
4. For each \(z \in N(x)\) do:
5. If \(HamPath(G - \{x\}, z, y)\), return true.
6. Return false

a. Explain (informally) why this algorithm is correct. (5 points) b. If every node of the graph G has degree (number of neighbors) at most 3, how long will this algorithm take at most? (15 points) (Hint: you can get a tighter bound than the most obvious one.)

We want to find a Hamiltonian path from x to y. The back-tracking algorithm branches on the first node visited after x. The choices are all the neighbors of x in the graph. We can’t return to x because the path must...
be simple. If we can find a Hamiltonian path from some neighbor \( z \) of \( x \) to \( y \) in \( G - \{x\} \), we can append \( x \) to the front of the path and get a Hamiltonian path from \( x \) to \( y \). On the other hand, if we cannot for any neighbor \( z \) of \( x \), then we cannot find a Hamiltonian path from \( x \) to \( y \), since the second node along such a path must be such a \( z \).

The time of the algorithm is proportional to the number of recursive calls that get made, since each time through the loop makes one recursive call and then does \( O(1) \) work. The recursion tree for the program has depth \( n \), since each time step we delete exactly one node from \( G \). In general graphs, the time could be as much as \((n - 1)!\) since the first node could have \( n - 1 \) neighbors, each possibility for the second node \( n - 2 \) neighbors, and so on. However, if no node in \( G \) has more than 3 neighbors, an obvious upper bound for the total time is \( O(3^n) \), since no node in the recursion tree has more than 3 children, and the depth is \( n \).

In fact, since after the first step, we only call \( y \) right after one of its neighbors \( x \) has been deleted from the tree, we can see that the total size of the tree is at most \( 3 \times 2^{n-1} = O(2^n) \).

There is an even more subtle argument that improves this bound: A graph with \( n \) nodes none of which has more than 3 edges can have at most \( 3n/2 \) edges total, since each edge touches two nodes. Every time we branch in the recursion, i.e., \( x \) has more than one neighbor, we delete at least 2 edges from the graph. Thus, any path in the recursion tree can have at most \((3n/2)/2 = 3n/4\) places where there was any branching. Thus, the time is actually bounded by \( O(2^{75n}) \) for such graphs.

**Implementation: 20 pts** Implement a back-tracking algorithm for maximum independent set (such as from class). Run your algorithm on random graphs with edge probability 1/2 (as in the previous assignment) for \( n \) as many different powers of 2 as you can without using more than a day computer time on any one instance. How does the actual maximum independent set size compare to the size found by the greedy heuristic last assignment?

The maximum independent set in a random graph approaches \( 2 \log n \) as \( n \) grows. So the independent set you find should be roughly twice as large as the one from the previous assignment. However, for small \( n \), the variance is high, so you might not be getting exactly this figure.